Robert Dudziński 293103
TKOM – Dokumentacja wstępna
Temat: O3 - niezmienniki pętli

Celem projektu jest stworzenie programu, który w pierwszej kolejności będzie weryfikował poprawność kodu wejściowego napisanego w niżej zdefiniowanym języku, a następnie będzie go optymalizował poprzez wykrywanie niezmienników i przenoszenie ich przed pętlę, w której się znajdują (wraz z obsługą zagnieżdżonych pętli).

WYMAGANIA:

- odczytanie kodu źródłowego z pliku
- analiza leksykalna, składniowa oraz semantyczna kodu wejściowego
- zgłaszanie informacji o pierwszym napotkanym błędzie podczas analiz (wraz z podaniem, w którym miejscu on występuje i prostym opisem przyczyny błędu)
- kod po optymalizacji musi zwracać te same rezultaty jak przed optymalizacją
- wyświetlenie rezultatu na wyjściu standardowym

Język będzie obsługiwał:

- warunki if else
- pętle while, for
- zmienne globalne i lokalne (zmienne lokalne deklarowane w danym bloku przesłaniają zmienne lokalne o tym samym identyfikatorze zadeklarowane w blokach nadrzędnych)
- wyrażenia arytmetyczne i logiczne
- deklaracje zmiennych o stałym typie
- obsługa typów int, float, bool oraz ich jednowymiarowych tablic

Przyjęte założenia:

- całe wejście dla programu musi znajdować się w jednym pliku
- plik zawiera jedną bezargumentową funkcję void main(), która w swojej definicji będzie zawierać cały kod, który będzie poddany analizie (dopuszcza się deklarację zmiennych globalnych)
- w kodzie nie ma możliwości korzystania ze wskaźników i operacji na fragmentach pamięci
- w przypadku wyrażeń logicznych wartości typu int oraz float równe 0 będą traktowane jako fałsz, a każda inna wartość będzie traktowana jako prawda
- w przypadku wyrażeń arytmetycznych zmienne typu bool w przypadku fałszu będą traktowane jako 0, natomiast w przeciwnym wypadku będą traktowane jako 1
- przypisanie wartości przy inicjalizacji tablicy skutkuje tym, że każdy jej element będzie miał taką wartość np. „int tab[4] = 2;” sprawi, że każdy element tablicy tab będzie miał wartość równą 2
- jeżeli przy inicjalizacji zmiennej nie przypisano jej wartości to domyślnie będzie przypisywana wartość 0
- język nie będzie obsługiwał prefixowych i postfixowych operatorów tj. „++” oraz „--„

Lista zdefiniowanych tokenów:

“void”	“(“, “)”	“{“	“}”	“+”	“-“	“*”
“/“	“;”	“if”	“else”	“while”	“for”	“&&”
“||”	“//“	“,”	“<“	“>”	“<=“	“>=“
“==“	“!=“	“=“	“!”	“int”	“float”	“bool
“true”	“false”	“return”	“continue”	“break”	“[“	“]”

GRAMATYKA

Uwaga: W celu czytelniejszego przedstawienia gramatyki symbol złączenia ‘,’ oznacza zapis ‘, { whiteChar } , ‘ gdzie whiteChar to zbiór białych znaków. Odstępstwem od tej zasady są symbole złączenia użyte przy definiowaniu id, varValue oraz intValue.

```
program = { initVar } , defFunction ;
defFunction = “void” , id , “(“ , “)” , block ;
block = ( “{“ , { singleStatement | block } , “}” ) | singleStatement ;
singleStatement = (conditionStatement | loopStatement ) |
                  ( ( initVar | return | break | continue | expression ) , “;” ) ;

return = “return” ;
break = “break” ;
continue = “continue” ;

conditionStatement = “if” , “(“ , logicalStatement , “)” , block , [ “else” , block]
;
loopStatement = ( “while” , “(“ , logicalStatement , “)” , block ) |
(“for” , “(“ , expression , “;” , logicalStatement , “;” , ( expression | “”) , “)”
, block ) ;

logicalStatement = andCondition , { “||” , andCondition } ;
andCondition = equalCondition , { “&&” , equalCondition } ;
equalCondition = relationalCondition , { equalOperator , relationalCondition } ;
relationalCondition = logicalParam , { relationalOperator , logicalParam } ;
logicalParam = [ “!” ] , ( logicalVal | var | ( “(“ , expression , “)” ) ) ;

equalOperator = “==“ | “!=“ ;
relationalOperator = “<“ | “<=“ | “>=“ | “>” ;

initVar = varType , var , [ “=“ , varValue ] ;
assignVar = var , “=“ , expression ;

expression = ( multiExpression , { addOperator  , multiExpression } ) |
logicalStatement ;
multiExpression = multiParam , { multiOperator , multiParam } ;
multiParam = var | ( “(“ , expression | assignVar , “)” ) | varValue ;

multiOperator = “*” | “/” ;
addOperator = “+” | “-“ ;

id = alphaChar , { alphaNumChar } ;
var = id , [ index ] ;
index = “[“ , intValue , “]” ;
varType = “int” | “float” | “bool” ;
varValue = logicalVal | ( [ “-“ ] , (
           ( digitChar , [ “.” [ { digitChar } ] ] ) |
           ( nonZeroChar , { digitChar } , [ “.” [ { digitChar } ] ] ) |
           ( “.” [ { digitChar } ] )
           ) ) ;
intValue = nonZeroChar , { digitChar } ;

logicalVal = “false” | “true” ;
alphaNumChar = alphaChar | digitChar ;
```

```
alphaChar = ? duże i małe znaki alfabetu angielskiego oraz znak ‘_’ ? ;
digitChar = ‘0’ | nonZeroChar ;
nonZeroChar = ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ ;
```

BUDOWA PROGRAMU (PODZIAŁ NA MODUŁY):

Obsługa plików
Oddzielny moduł będzie odpowiedzialny za obsługę plików. Będzie on posiadał metody do zwracania kolejnych znaków z ustalonego pliku jak również będzie on mógł obsługiwać wyjście do piku. W celu optymalizacji operacji na plikach moduł będzie np. buforował odczytywany i zapisywany tekst, aby nie wywoływać przerwań systemowych w celu odczytania pojedynczych znaków (zamiast tego moduł wczyta od razu cały plik tekstowy).

Analizator leksykalny
Lekser będzie na swoim wejściu dostał kolejne znaki z pliku wejściowego i na tej podstawie będzie tworzył kolejne tokeny, które będą przekazywane do analizatora składniowego. Analizator leksykalny będzie również obsługiwał komentarze (pojedynczej linii “//“ jak również komentarz zakresu “/* (…) */“) i od razu będzie on usuwany. Analiza leksykalna będzie oparta na automacie. Białe znaki będą pomijane na etapie analizy leksykalnej.

Analizator składniowy
Analizator składniowy będzie na wejściu otrzymywał kolejne tokeny od analizatora leksykalnego. Na tej podstawie będzie dokonywał rozbioru wyrażeń w celu sprawdzenia czy otrzymane tokeny zgadzają się ze zdefiniowaną wyżej gramatyką programu.

Analizator semantyczny
Moduł będzie sprawdzał poprawność znaczenia rozbioru składniowego otrzymanego od poprzedniego modułu. Ze względu na zastosowane w założeniach typy zmiennych (int, float oraz bool) jest możliwe obsłużenie wyrażeń arytmetycznych oraz logicznych zawierających w sobie odwołania do zmiennych dowolnego typu (pod warunkiem, że nie jest to odwołanie do zmiennej reprezentującej całą tablicę (a nie pojedynczy element) – w takim przypadku moduł zgłosi błąd).

Optymalizator
Praca optymalizatora będzie się opierać na zapisywaniu w generowanej tablicy symboli widoczności zmiennej w każdym bloku oraz na sprawdzaniu czy w danym bloku (który należy do pętli) są wykonywane operacje mogące zmodyfikować wartość zmiennej lub czy operacje mogące zmieniać wartość ustawiają tą wartość na podstawie stałych lub innych zmiennych, które w ramach danego bloku mają stałą wartość.

Obsługa błędów
Każdy z modułów będzie mógł się komunikować z modułem do obsługi błędów. W przypadku napotkania błędu przez któryś z modułów dalsza analiza i/lub proces optymalizacji zostanie przerwany i zostanie wyświetlony stosowny komunikat.

STRUKTURY DANYCH:

Tablica symboli
Podczas przygotowywania kodu do procesu optymalizacji będzie generowana tablica symboli. Każdy z elementów tablicy będzie zawierał w sobie informacje o pojedynczej zmiennej. Na rekord pojedynczej zmiennej będzie się składał identyfikator tej zmiennej, a ponieważ program obsługuje przesłanianie zmiennych (poprzez deklarację zmiennych o tej samej nazwie w bardziej zagnieżdżonych blokach) to rekord będzie zawierał w sobie pole przechowujące poziom zagnieżdżenia bloku, w którym dana zmienna była zadeklarowana tj. zmienne globalne mają poziom 0, zmienne zadeklarowane bezpośrednio w bloku funkcji main() mają poziom 1 itd..

Do każdego rekordu będzie dołączona lista przechowująca informację o miejscu pierwszego odczytu, pierwszego zapisu jak również ilości zapisów do zmiennej w danym bloku. Każdy element listy będzie opisywał pojedynczy blok (więcej informacji w sekcji opisującej algorytm optymalizacji).

Drzewo rozbioru
Podczas analizy składniowej moduł będzie tworzył na podstawie tokenów drzewo rozbioru w celu sprawdzenia zdefiniowanej w programie gramatyki. Każdy z węzłów drzewa będzie w sobie zawierał informację o symbolu jaki reprezentuje. Korzeniem drzewa będzie symbol oznaczający cały kod programu.

Tablica zdefiniowanych tokenów
Podczas analizy leksykalnej będzie potrzebna informacja o symbolach zarezerwowanych podczas generowania tokenów przez lekser.

INNE INFORMACJE:

W celu przyśpieszenia weryfikacji poprawności kodu, lekser będzie od razu po wygenerowaniu tokenu przekazywał go do analizatora składniowego, żeby nie weryfikować niepotrzebnie poprawności leksykalnej pozostałego kodu w przypadku, gdy do tego momentu był znaleziony np. błąd gramatyczny.

Program
Program będzie aplikacją konsolową napisaną w języku wysokiego poziomu (C++ lub Java). Przy jego wywołaniu będzie podana ścieżka do pliku wejściowego. Rezultat będzie wyświetlony na standardowym wyjściu.

Testowanie
W celu testowania zostaną napisane testy jednostkowe (w zależności od wybranego języka programowania będzie to najpewniej JUnit w przypadku Javy lub Boost w przypadku C++). Testy będą sprawdzać np. czy analizator leksykalny prawidłowo wykrywa tokeny na podstawie danego wejściowego ciągu znaków oraz czy analizator składniowy prawidłowo weryfikuje zdefiniowaną gramatykę.

WSTĘPNA KONCEPCJA ALGORYTMU OPTYMALIZACJI

Każda zmienna z tablicy symboli będzie przechowywała listę bloków, w której jest ona widoczna, czyli blok, w którym została zadeklarowana dana zmienna oraz wszystkie zagnieżdżone bloki wewnątrz bloku z deklaracją (wliczając to bloki, które nie są "bezpośrednio" zagnieżdżone, czyli co raz głębsze bloki). Ponieważ musimy potrafić rozróżnić każdy z bloków musimy mu przypisać na jakiej "głębokości zagnieżdżenia" się znajduje względem bloku deklaracji oraz który jest to blok z kolei odliczając bloki w bloku nadrzędnym (wliczając w to jedynie bloki o głębokości równej głębokości sprawdzanego bloku).

1. Dla każdego bloku, w którym może występować zmienna musimy sprawdzić, ile jest tam operacji, które mogą zmodyfikować wartość danej zmiennej, czyli ile jest operacji przypisania (wliczając w to przypisania w blokach zagnieżdżonych). Jeżeli liczba takich operacji jest większa niż jeden, to rozważana zmienna nie może być wyciągnięta poza obecny blok (możliwy jest wyjątek w sytuacji, w której wszystkie operacje przypisania przypisują taką samą stałą lub zmienną, która w danym bloku ma stałą wartość). W przypadku, kiedy jest tylko jedna operacja przypisania, to ta zmienna jest podejrzana o możliwość optymalizacji.

2. W celu sprawdzenia czy rzeczywiście możemy przerzucić daną operację przypisania przed aktualny blok, musimy sprawdzić dwa warunki:

2.1. Czy podczas tego jedynego przypisania jest przypisana wartość stała lub wartość wyrażenia opartego na zmiennych, których wartość jest stała w ramach tego bloku (możemy to odczytać również na podstawie ilości przypisań tej drugiej zmiennej w rozważanym bloku).

2.2. Czy między początkiem bloku, a tym jedynym przypisaniem, jest jakaś operacja, która odwoływałaby się do tej zmiennej. W tym celu w trakcie tworzenia tablicy symboli musimy zapisać w niej również miejsce, w którym jest pierwsze odwołanie do zmiennej w tym bloku oraz miejsce, w którym znajduje się pierwsze przypisanie wartości do tej zmiennej. Mając te dwie dane możemy porównać, która z tym operacji nastąpiła jako pierwsza.

3. Jeżeli oba powyższe warunki są spełnione, to możemy tą jedyną operację przypisania wartości do zmiennej wyciągnąć przed obecny blok.

3.1. W przypadku, jeśli przenieśliśmy przypisanie, które znajdowało się wewnątrz innego wyrażenia, to w miejscu tego przypisania możemy wstawić samą zmienną.

Przykłady optymalizacji:

Przed:

```
int a = 5;  int b = 10; int c = 20;
int d = 1;  int e = 2;  int f = 3;
int g = 4;  int h = 6;

void main()
{
    for (int i = 0; i < 5; i = i + 1)
    {
        int m = 5;
        h = 10 * a;
        d = b * 7 + 15;
        f = b * 7 + 15;
        c = 50;
        g = 20;
        int j = 10;
        while (j > 10)
        {
            j = j - 1;
            g = 30;
            e = 2 * c;
            for (int k = 0; k < 100; k = k + 1)
            {
                m = (i - 5) * 5;
                f = 20;
                if (f > 15)
                    a = 8;
            }
            f = 1;
        }
    }
}
```

Po:

```
int a = 5;  int b = 10; int c = 20;
int d = 1;  int e = 2;  int f = 3;
int g = 4;  int h = 6;

void main()
{
    d = b * 7 + 15;
    c = 50;
    e = 2 * c;          //adnotacja 1
    for (int i = 0; i < 5; i = i + 1)
    {
        int m = 5;
        h = 10 * a;     //adnotacja 2
        f = b * 7 + 15; //adnotacja 3
        g = 20;
        int j = 10;
        g = 30;
        m = (i - 5) * 5;//adnotacja 4
        while (j > 10)
        {
            j = j - 1;
            f = 20;     //adnotacja 5
            for (int k = 0; k < 100; k = k + 1)
            {
                if (f > 15)
                    a = 8;
            }
            f = 1;
        }
    }
}
```

Adnotacja 1: przeniesione dwa bloki (pętle) wyżej
Adnotacja 2: nie można przenieść, ponieważ a może zmienić wartość
Adnotacja 3: nie można przenieść, ponieważ f może zmienić wartość
Adnotacja 4: przenieśliśmy, ponieważ w obszarze, w którym żyje zmienna m, i ma stałą wartość
Adnotacja 5: przeniesione tylko jeden blok wyżej, bo w obecnym bloku f może zmienić wartość

Uwaga: Kod ma zadanie przedstawić ideę optymalizacji zmiennych. Zakładamy, że w ten kod mogą być „wplecione” inne operacje korzystające z tych zmiennych, dlatego np. nie usuwamy „g = 20;” i nie wyciągamy „g = 30;” przed główną pętlę.

Przykłady informowania użytkownika o znalezionych błędach:

```
void main()
{
    abc = 3;
}
```

```
    abc = 3;
    ^
line 3: cannot regognize symbol
```

```
void main()
{
    int tab[20];

    int a = 3 * tab;
}
```

```
    int a = 3 * tab;
                ^
line 5: invalid variable type used in expression
```

```
void main()
{
    float a;
    float b;
    float c = a b;
}
```

```
    float c = a b;
                ^
line 5: unexpected symbol
```

Przykłady optymalizacji z mniejszą liczbą zmiennych:

```
float a = 5;
void main()
{
    for (int i = 0; i < 30; i = i + 1)
    {
        int d = a / 2;
        a = 3;
        //kod niemodyfikujący a
    }
}
```

```
float a = 5;
void main()
{
    for (int i = 0; i < 30; i = i + 1)
    {
        int d = a / 2;
        a = 3; //nie możemy przenieść zmiennej, bo wcześniej jest odczytywana wartość zmiennej a
        //kod niemodyfikujący a
    }
}
```

```
float a = 5;
void main()
{
    for (int i = 0; i < 30; i = i + 1)
    {
        //kod nieodczytujący a
        a = 3;
        int d = a / 2;
        //kod niemodyfikujący a
    }
}
```

```
float a = 5;
void main()
{
    a = 3;
    for (int i = 0; i < 30; i = i + 1)
    {
        //kod nieodczytujący a
        int d = a / 2;
        //kod niemodyfikujący a
    }
}
```

```
int a = 3;
void main()
{
    for (int i = 0; i < 20; i = i + 1)
    {
        a = 6;
        //kod niemodyfikujący a
        a = 4;
    }
}
```

```
int a = 3;
void main()
{
    for (int i = 0; i < 20; i = i + 1)
    {
        a = 6; //nie możemy przenieść zmiennej, bo później jest zmieniana wartość zmiennej a
        //kod niemodyfikujący a
        a = 4;
    }
}
```

```
int a = 3;

void main()
{
    for (int i = 0; i < 20; i = i + 1)
    {
        //kod nieodczytujący a
        a = 6;
        int a = 8;
        //kod niemodyfikujący a
        a = 4;
    }
}
```

```
int a = 3;

void main()
{
    a = 6; //możemy przenieść, bo później jest modyfikowana inna zmienna (o tym samym identyfikatorze)
    for (int i = 0; i < 20; i = i + 1)
    {
        //kod nieodczytujący a
        int a = 8;
        //kod niemodyfikujący a
        a = 4;
    }
}
```

```
int a = 3;

void main()
{
    for (int i = 0; i < 20; i = i + 1)
    {
        //kod nieodczytujący a
        int e = 20 + (a=4);
        //kod niemodyfikujący a
    }
}
```

```
int a = 3;

void main()
{
    a=4;
    for (int i = 0; i < 20; i = i + 1)
    {
        //kod nieodczytujący a
        int e = 20 + (a);
        //kod niemodyfikujący a
    }
}
```

```
void main()
{
    for (int i = 0; i < 20; i = i + 1)
    {
        int k;
        for (int j = 0; j < 10; j = j + 1)
        {
            //kod nieodczytujący k
            k = i;
            //kod niemodyfikujący k
        }
    }
}
```

```
void main()
{
    for (int i = 0; i < 20; i = i + 1)
    {
        int k;
        k = i;
        for (int j = 0; j < 10; j = j + 1)
        {
            //kod nieodczytujący k
            //kod niemodyfikujący k
        }
    }
}
```